A Hierarchical Variability Model for Software Product Lines*

Dilian Gurov[1], Bjarte M. Østvold[2], and Ina Schaefer[3]

[1] Royal Institute of Technology, Stockholm, Sweden
dilian@csc.kth.se
[2] Norwegian Computing Center, Oslo, Norway
bjarte@nr.no
[3] TU Braunschweig, Germany
i.schaefer@tu-bs.de

Abstract. A key challenge in software product line engineering is to represent solution space variability in an economic, yet easily understandable fashion. We introduce the notion of hierarchical variability models to describe families of products in a manner that facilitates their modular design and analysis. In this model, a family is represented by a *common* set of artifacts and a set of *variation points* with associated variants. A variant is again a hierarchical variability model, leading to a hierarchical structure. These models, however, are not unique with respect to the families they define. We therefore propose a quantitative measure on hierarchical variability models that expresses the degree to which a variability model captures commonality and variability in a family. Further, by imposing well-formedness constraints, we identify a class of variability models that, by construction, have maximal measure and are unique for the families they define. For this class of *simple families*, we provide a procedure that reconstructs their hierarchical variability model. The reconstructed model can be used to drive various static analyses by divide-and-conquer reasoning. Hierarchical variability models strike a balance between the formalism's expressiveness and the desirable property of model uniqueness. We illustrate the approach by a small product line of Java classes.

1 Introduction

System diversity is prevalent in modern software systems. Systems simultaneously exist in many different variants in order to comply with different requirements. Software product line engineering [18] aims at developing a family of system variants by managed reuse in order to decrease time to market and to improve quality. The variability of the different products in a software product line can be represented at different levels [7]. *Problem space variability* describes

* Partly funded by the EU project *HATS*, Highly Adaptable and Trustworthy Software using Formal Models (FP7-231620) and the Deutsche Forschungsgemeinschaft (SCHA1635/2-1).

R. Hähnle et al. (Eds.): ISoLA 2011 Workshops, CCIS 336, pp. 181–199, 2012.

product variation in terms of features where a feature is a user-visible product characteristic. The set of valid feature configurations defines the set of possible products. However, features do not relate to the actual artifacts that are used to realize the products.

Problem-space variability, based on features, is at the requirements level, while *solution space variability* is at the design and implementation level. Solution-space variability describes product variation in terms of shared artifacts that are used to build the actual products of the product line. In this paper, we capture solution space variability in terms of variable artifact implementations for fixed artifact names. This means that in different product variants an artifact with the same name can be realized with different implementations. Here, an artifact can be a component at a suitable level of granularity, such as a method, a class, or a module. Then, the artifact name would be the method signature (including the method name), interface signature or module signature, respectively, while the artifact implementation would be the method body, interface implementation, or the module realization. Previously, we used the finest of these levels [20], i.e., method signatures and method bodies, while in Section 4 we show another interpretation, where artifact names are types and artifact implementations are classes, interfaces or types implementing the former type.

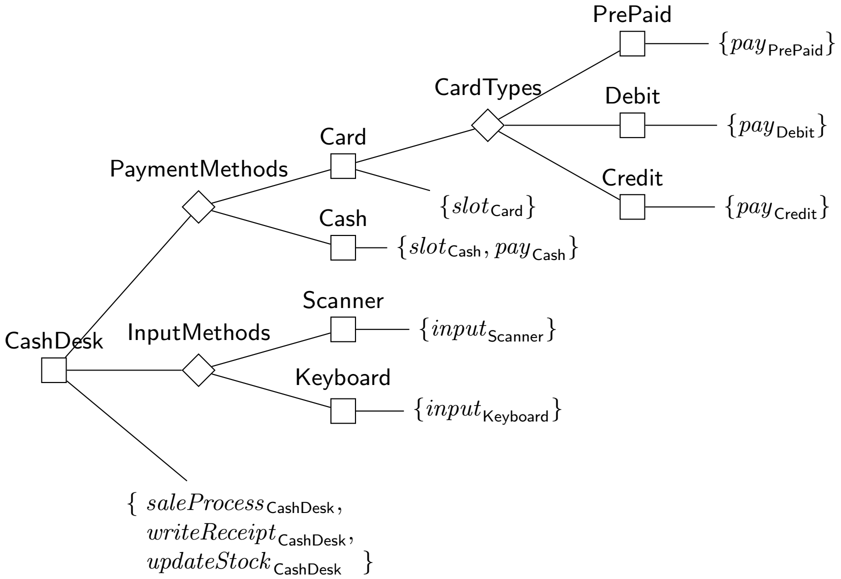
In order to describe the relationship of the artifact names to the artifact implementations in the product variants, we introduce *hierarchical variability models*. Hierarchical variability models represent, in a hierarchical manner, the artifact implementations that are common to all products and the variations in the artifact implementations that can occur between different products. On each hierarchical level, there is a *common set* of artifact implementations that represent parts shared by all products, while *variation points* represent parts that can vary from product to product. Every variation point is associated with a set of variants that represent choices for realizing the variation point in different ways. A variant is itself represented by a hierarchical variability model, introducing a new level of hierarchy. A product described by a hierarchical variability model is obtained by selecting a variant at every variation point.

We have previously argued that hierarchical variability models support modular design [12] and divide-and-conquer reasoning for product lines, such as the formal verification of critical requirements of all products of a family [20]. In general, given a concrete program analysis, factoring out common implementations naturally reduces redundancy in the analysis. At variants with more than one variation point, the analysis problem is *decomposed* into simpler subproblems (since they expose orthogonality), while at variation points with more than one variant, the same problem is solved *independently* as a case analysis (since they don't share implementations). Thus, a hierarchical variability model can be seen as a (divide-and-conquer style) scheme for decomposing and splitting an analysis over a family.

In this paper, we propose a hierarchical variability model that is *simple* in the sense that it requires the choice of exactly one variant for every variation point, and does not specify any constraints between choices made at different variation points. Figure 1 shows a simple hierarchical variability model for a cash desk system, depicted as a tree with a root node marked CashDesk. Common to all cash desk systems are the following artifact implementations: $saleProcess_{\mathsf{CashDesk}}$ for handling the sale process, and two implementations called $writeReceipt_{\mathsf{CashDesk}}$ and $updateStock_{\mathsf{CashDesk}}$ responsible for the corresponding tasks. The notational convention is that an artifact implementation is an artifact name (*e.g.*, *saleProcess*) with an index (*e.g.*, CashDesk).

At the first level of hierarchy, a cash desk can vary in two uncorrelated (*i.e.*, orthogonal) aspects. First, there are two methods to input data about merchandise payed for at the cash desk: by keyboard or using a scanner. Second, there are two ways to pay, either in cash or by card. Thus, on the first level, the hierarchical variability model in the figure has two variation points: InputMethods and PaymentMethods. Each variation point has associated variants which capture one particular way of realizing the variation point. Variation point InputMethods has two variants, Keyboard and Scanner, each with an implementation of the corresponding input method. Variation point PaymentMethods also has two variants: Cash and Card for the two forms of payment. Both variants provide an artifact named *slot* for inserting the means of payment and *pay* for the actual payment process with different implementations. *slot* has one implementation in each variant, whereas *pay* has one implementation for cash and three variant implementations for card, corresponding to three different types of card.

The intention of a hierarchical variability model is that, on each level of hierarchy, *common sets* of artifact implementations are factored out, while *uncorre-*


PrePaid
$\{pay_{\mathsf{PrePaid}}\}$
CardTypes
Debit
$\{pay_{\mathsf{Debit}}\}$
Card
PaymentMethods
Credit
$\{slot_{\mathsf{Card}}\}$
$\{pay_{\mathsf{Credit}}\}$
Cash
$\{slot_{\mathsf{Cash}}, pay_{\mathsf{Cash}}\}$
Scanner
CashDesk
InputMethods
$\{input_{\mathsf{Scanner}}\}$
Keyboard
$\{input_{\mathsf{Keyboard}}\}$
$\{saleProcess_{\mathsf{CashDesk}},$
$writeReceipt_{\mathsf{CashDesk}},$
$updateStock_{\mathsf{CashDesk}}\}$

Legend: □ root of SHVM; ◇ variation point.

Fig. 1. The CashDesk hierarchical variability model (drawn sideways)

lated (or *orthogonal*) sets of artifact implementations are delegated to different variation points. To provide a measure for the quality of hierarchical variability models for defining a family in an economical way, we define the *separation degree* of a model (Definition 6) as the ratio between the total number of artifact implementations from which products are constructed and the total number of artifact implementation occurrences in the common sets of the model. Thus, high-quality models capture repetitions across products in a family without repetition in the model. The maximal possible separation degree of one is reached in models where every artifact implementation occurs in exactly one common set.

In order to reason formally about hierarchical variability models, we provide these with a formal semantics in terms of the products that can be generated by variant selection. We define *well-formedness constraints* on hierarchical variability models, under which the separation degree of the model is equal to one by construction. We term the class of families generated by well-formed variability models *simple families*, and define this class in a model-independent fashion. We present a transformation from simple families to hierarchical variability models that (re)constructs the unique well-formed model that generates the family. Uniqueness is established by showing that the two transformations—from well-formed models to simple families and *vice versa*—are inverses of each other. For practical purposes, the latter transformation can be used for *variability model mining* from a given family of products. Simple hierarchical variability models thus strike a balance between the expressiveness of the modeling formalism—no bindings and being grammar-like—and the desirable property of uniqueness of models: With a more expressive modeling formalism uniqueness may not be achievable.

To the best of our knowledge, this work is the first to provide a formal semantics for hierarchical variability models in the solution space, and to characterize a class of variability models through the class of generated product families. Previous work has been informal, as for instance the Koala component model [22], hierarchical variability modeling for software architectures [12], or plastic partial components [17]. Our work is also the first to provide a technique for constructing a hierarchical solution space variability model from a given family.

Our main *contributions* are thus:

(i) A formal definition of *simple families* as families that can be formed from artifact implementations by using a set of base operations on families (Section 2.1).

(ii) A definition of *simple hierarchical variability models*, together with a quality measure called *separation degree* and a set of *well-formedness constraints* yielding (by construction) models with maximal measure (Section 2.2).

(iii) A formal semantics for hierarchical variability models in terms of *family generation*, and a proof that, for every well-formed variability model, the generated family is simple (Section 3.1).

(iv) A procedure to construct hierarchical variability models from simple families that produces well-formed models (Section 3.2).

(v) A *characterization result* stating that, for well-formed hierarchical variability models and simple families, family generation and hierarchical variability model construction are *inverses* of each other, thus implying correctness of model construction and uniqueness of well-formed models with respect to the families they generate (Section 3.3).

All proofs and some supporting results can be found in the accompanying technical report [11].

2 Families and Variability Models

In this section, we present product families as a semantic domain for our hierarchical variability model. The model is presented in the following subsection.

2.1 Families

We consider products realized by a set of artifact implementations for a given set of artifact names. An artifact can be thought of as, *e.g.*, a component or a method. We fix a countably infinite set of artifact names Art.

Definition 1 (Product, family). *An* artifact implementation *is an indexed artifact name; let a_i denote the i-th implementation of artifact name a. A* product *P is a finite set of artifact implementations, where for each artifact name there is at most one implementation. A* family *$\mathcal{F}$ is a finite non-empty set of products.*

Thus, products can be seen as partial maps from artifact names to natural numbers, having a finite domain; we use Nat^{Art} to denote the set of all products over Art. We refer to singleton set families as *core* families, or simply cores. The family consisting of the empty product is denoted $1_{\mathcal{F}}$.

Example 1. Here are some families that are used later to illustrate various notions.

$$\begin{aligned}\mathcal{F}_A = \{ \ &\{a_1, b_1, c_1, d_1, e_1\}, \{a_1, b_1, c_1, d_1, e_2\}, \{a_1, b_1, c_2, d_2, e_1\},\\ &\{a_1, b_1, c_2, d_2, e_2\}, \{a_1, b_1, c_2, d_3, e_1\}, \{a_1, b_1, c_2, d_3, e_2\} \ \}\\ \mathcal{F}_B = \{ \ &\{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_1\} \ \}\end{aligned}$$

Next, we define two mappings for identifying the artifact names and artifact implementations that occur in a family.

Definition 2 (Family names and implementations). *The mapping* names$(\mathcal{F})$ *from families to sets of artifact names and the mapping* impls$(\mathcal{F})$ *from families to sets of artifact implementations are defined as follows, where $a^1, \ldots, a^n \in Art$ and $i_1, \ldots, i_n \in Nat$:*

$$\begin{aligned}\text{names}(\mathcal{F}) &\overset{\mathsf{def}}{=} \textstyle\bigcup_{P\in\mathcal{F}} \text{names}(P)\\ \textit{where } \text{names}\left(\{a^1_{i_1}, \ldots, a^n_{i_n}\}\right) &\overset{\mathsf{def}}{=} \{a^1, \ldots, a^n\}\\ \text{impls}(\mathcal{F}) &\overset{\mathsf{def}}{=} \textstyle\bigcup_{P\in\mathcal{F}} \text{impls}(P)\\ \textit{where } \text{impls}(\{a^1_{i_1}, \ldots, a^n_{i_n}\}) &\overset{\mathsf{def}}{=} \{a^1_{i_1}, \ldots, a^n_{i_n}\}\end{aligned}$$

In this definition we abuse notation by also defining mappings with the same names from products to the same co-domains.

We use two binary operations on families, the usual set union operation $\cup$ and the *product union* operation $\bowtie$ over families with disjoint sets of artifact names defined by:

$$\mathcal{F}_1 \bowtie \mathcal{F}_2 \stackrel{\mathsf{def}}{=} \{P_1 \cup P_2 \,|\, P_1 \in \mathcal{F}_1 \wedge P_2 \in \mathcal{F}_2\}$$

and generalized through $\prod_{i \in I} \mathcal{F}_i$ to non-empty sets of families. Intuitively, the product union of two families is the family having as products all possible combinations of products of the original families. Both operations are commutative and associative.

We now define a distinct class of families that we later relate to a specific class of hierarchical variability models. The class of families contains all single-product families consisting of a single artifact implementation, and is closed under product union of families over disjoint sets of artifact names, and under union of families over the same set of artifact names, but having disjoint implementations.

Definition 3 (Simple family). *The class $\boldsymbol{F}$ of* simple *families is the least set of families closed under the formation rules:*

($\mathcal{F}$1) $\{\{a_i\}\} \in \boldsymbol{F}$ *for any $a \in Art$ and $i \in Nat$.*
($\mathcal{F}$2) $\mathcal{F}_1 \bowtie \mathcal{F}_2 \in \boldsymbol{F}$ *for any $\mathcal{F}_1, \mathcal{F}_2 \in \boldsymbol{F}$ such that* $\mathrm{names}\,(\mathcal{F}_1) \cap \mathrm{names}\,(\mathcal{F}_2) = \emptyset$.
($\mathcal{F}$3) $\mathcal{F}_1 \cup \mathcal{F}_2 \in \boldsymbol{F}$ *for any $\mathcal{F}_1, \mathcal{F}_2 \in \boldsymbol{F}$ such that* $\mathrm{names}\,(\mathcal{F}_1) = \mathrm{names}\,(\mathcal{F}_2)$ *and* $\mathrm{impls}(\mathcal{F}_1) \cap \mathrm{impls}(\mathcal{F}_2) = \emptyset$.

Example 2. The family $\{\,\{a_1, b_1\}, \{a_1, b_2\}\,\}$ is simple, as it can be presented as $\{\{a_1\}\} \bowtie (\{\{b_1\}\} \cup \{\{b_2\}\})$ which follows the above formation rules. Family $\mathcal{F}_A$ of Example 1 is also simple (as we shall see later in Example 6), while family $\mathcal{F}_B$ of Example 1 is not: there is no way of building this family with the above formation rules.

Simplicity of families expresses that different functionalities in a product line are always orthogonal, and that alternative realizations of the same functionality have always disjoint implementations. These assumptions are rather heavy and may not always hold in practice. But only under such severe constraints can one hope for such a (strong) uniqueness result as the one obtained later (Section 2.1).

Two distinct artifact names $a, b \in \mathrm{names}\,(\mathcal{F})$ are termed *correlated* in a family $\mathcal{F}$, denoted $aC_{\mathcal{F}}b$, if there are implementations $a_i, b_j \in \mathrm{impls}(\mathcal{F})$ such that no product in $\mathcal{F}$ contains both implementations simultaneously. Otherwise, names a and b are termed *uncorrelated* or *orthogonal*. The correlation relation $C_{\mathcal{F}}$ on $\mathrm{names}\,(\mathcal{F})$ is symmetric, and hence, its reflexive and transitive closure $C^*_{\mathcal{F}}$ is an equivalence relation. As usual, we denote the partitioning induced by $C^*_{\mathcal{F}}$ on $\mathrm{names}\,(\mathcal{F})$ by $\mathrm{names}\,(\mathcal{F})\,/C^*_{\mathcal{F}}$ (quotient set).

Example 3. Consider family $\mathcal{F}_A$ of Example 1. The only two correlated names are c and d, evidenced by the lack of a product containing, for instance, c_1 and d_2. Thus, we have $\mathrm{names}\,(\mathcal{F}_A)\,/C^*_{\mathcal{F}_A} = \{\{a\}, \{b\}, \{c, d\}, \{e\}\}$.

Correlation (and orthogonality) extends naturally to products in a family: Products P and P' are correlated in $\mathcal{F}$ if some artifact name occurring in P is correlated to some artifact name occurring in P'. Similarly, we define the sharing relation $N_{\mathcal{F}}$ on $\mathcal{F}$ as $P_1 \, N_{\mathcal{F}} \, P_2 \overset{\mathsf{def}}{\Leftrightarrow} P_1 \cap P_2 \neq \emptyset$, and use its reflexive and transitive closure $N^*_{\mathcal{F}}$ to partition the family $\mathcal{F}$.

When restricted to simple families, the two operations on families do not distribute over each other. This entails that simple families have *unique* formation trees modulo commutativity and associativity of the two operations associated with the rules.

2.2 Variability Models

In order to represent solution space variability of families in terms of shared artifact implementations, we consider simple hierarchical variability models.

Definition 4 (Simple hierarchical variability model). *A* simple hierarchical variability model (SHVM) $\mathcal{S}$ *is inductively defined as:*

(i) *a (possibly empty)* common set *of artifact implementations* M_C*, or*
(ii) *a pair* $(M_C, \{VP_1, \dots, VP_n\})$ *where* M_C *is defined as above and the set* $\{VP_1, \dots, VP_n\}$ *of* variation points *is non-empty. A variation point* $VP_i = \{\mathcal{S}_{i,j} \mid 1 \leq j \leq k_i\}$*, where* $k_i \geq 2$*, is a set of (at least two) SHVMs called* variants*.*

We sometimes refer to an SHVM simply as a variability model. An SHVM with only a common set of artifact implementations is called *ground model.* An SHVM generates a family $\mathcal{F}$ through all possible ways of resolving the variabilities of the SHVM. This process recursively selects exactly one variant for each variation point. We defer a formal definition of such a semantics for SHVMs to Section 3.1. Variability models can be naturally depicted as trees, where leaves are common sets of artifact implementations, and internal nodes are the roots of SHVMs or variation points.

Example 4. Figure 2 and Figure 3 show four variability models named $\mathcal{S}_{A1}$, $\mathcal{S}_{A2}$, $\mathcal{S}_{B1}$, and $\mathcal{S}_{B2}$. In these figures, (sub)trees showing variability models are rooted with boxes, and subtrees showing variation points are rooted with diamonds.

In analogy with Definition 2, we define two mappings for identifying the artifact names and artifact implementations that occur in SHVMs.

Definition 5 (SHVM names and implementations). *The mapping* names$(\mathcal{S})$ *from SHVMs to sets of artifact names and the mapping* impls$(\mathcal{S})$ *from SHVMs to sets of artifact implementations are defined as follows, where* $a^1, \dots, a^n \in Art$ *and* $i_1, \dots, i_n \in Nat$*:*

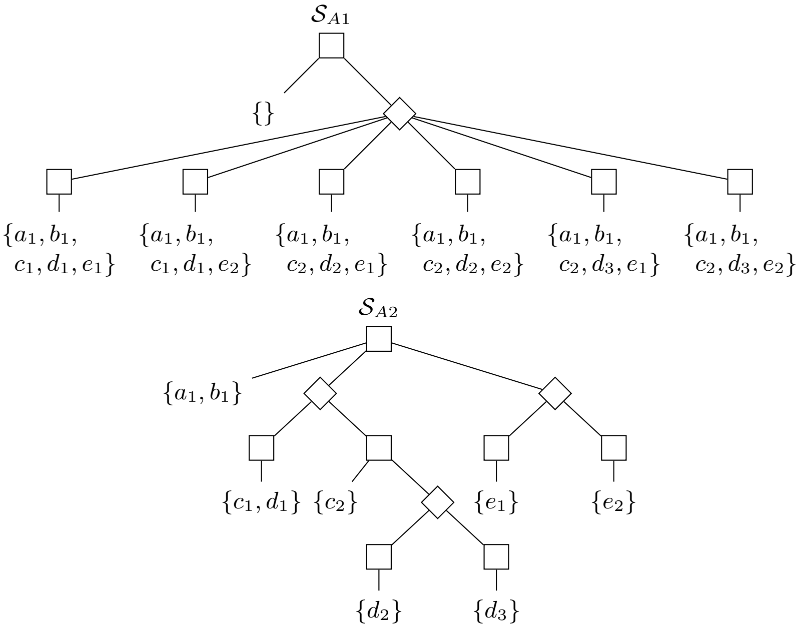

$\mathcal{S}_{A1}$
$\{\}$
$\{a_1, b_1, c_1, d_1, e_1\}$
$\{a_1, b_1, c_1, d_1, e_2\}$
$\{a_1, b_1, c_2, d_2, e_1\}$
$\{a_1, b_1, c_2, d_2, e_2\}$
$\{a_1, b_1, c_2, d_3, e_1\}$
$\{a_1, b_1, c_2, d_3, e_2\}$
$\mathcal{S}_{A2}$
$\{a_1, b_1\}$
$\{c_1, d_1\}$
$\{c_2\}$
$\{e_1\}$
$\{e_2\}$
$\{d_2\}$
$\{d_3\}$

Fig. 2. SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ for the family $\mathcal{F}_A$ in Example 1

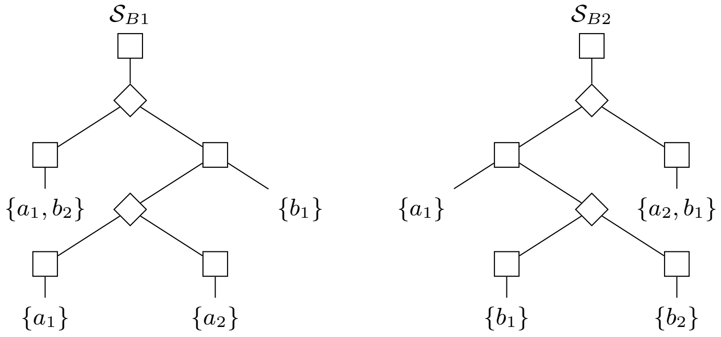


Fig. 3. SHVMs $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ for the family $\mathcal{F}_B$ in Example 1

$$\begin{aligned}
\text{names}\left(\{a^1_{i_1},\ldots,a^n_{i_n}\}\right) &\overset{\mathsf{def}}{=} \{a^1,\ldots,a^n\}\\
\text{names}\left((M_c,\{VP_1,\ldots,VP_n\})\right) &\overset{\mathsf{def}}{=} \text{names}\,(M_C)\cup\textstyle\bigcup_{1\leq i\leq n}\text{names}\,(VP_i)\\
\textit{where}\ \text{names}\,(VP) &\overset{\mathsf{def}}{=} \textstyle\bigcup_{\mathcal{S}\in VP}\text{names}\,(\mathcal{S})\\
\text{impls}(\{a^1_{i_1},\ldots,a^n_{i_n}\}) &\overset{\mathsf{def}}{=} \{a^1_{i_1},\ldots,a^n_{i_n}\}\\
\text{impls}((M_c,\{VP_1,\ldots,VP_n\})) &\overset{\mathsf{def}}{=} \text{impls}(M_C)\cup\textstyle\bigcup_{1\leq i\leq n}\text{impls}(VP_i)\\
\textit{where}\ \text{impls}(VP) &\overset{\mathsf{def}}{=} \textstyle\bigcup_{\mathcal{S}\in VP}\text{impls}(\mathcal{S})
\end{aligned}$$

Again we abuse notation by also defining mappings with the same names from variation points to the same co-domains.

Next, we define a measure of the degree of separation in a variability model, as the proportion between the number of artifact implementations of a variability model and the total size of the leaves of the SHVM tree. The separation degree is, thus, a number in the interval $\langle 0,1]$, and captures the degree to which the commonalities and orthogonalities of products are factored out as common sets and variation points in a variability model, respectively: the higher this degree, the less artifact implementations occur repeatedly in more than one leaf. The maximal value of 1 holds when every artifact implementation occurs in exactly one leaf; this is trivially the case for ground models.

Definition 6 (Separation degree). *The* separation degree $sd(\mathcal{S})$ *of a variability model* $\mathcal{S}$ *is defined as:*

$$\begin{aligned}
sd(\{\}) &\overset{\mathsf{def}}{=} 1\\
sd(\mathcal{S}) &\overset{\mathsf{def}}{=} \frac{|\text{impls}(\mathcal{S})|}{sd'(\mathcal{S})} \qquad \textit{if}\ \mathcal{S}\neq\{\}
\end{aligned}$$

where $sd'(\mathcal{S})$ *is inductively defined as follows:*

$$\begin{aligned}
sd'(M_C) &\overset{\mathsf{def}}{=} |M_C|\\
sd'((M_C,\{VP_1,\ldots,VP_n\})) &\overset{\mathsf{def}}{=} sd'(M_C)+\Sigma_{1\leq i\leq n}sd'(VP_i)\\
\textit{where}\ sd'(VP) &\overset{\mathsf{def}}{=} \Sigma_{\mathcal{S}\in VP}sd'(\mathcal{S})
\end{aligned}$$

As usual $|S|$ denotes the cardinality of set S.

Intuitively this definition captures the extent to which orthogonal artifact implementations are delegated to separate variation points, and the extent to which disjointness of artifact implementations is delegated to separate variants. Since this is the original intention of variation points and variants in our model, separation degree is an obvious quality measure indicating how well the model is used for the purpose of hierarchically representing a software family (that is, a set of products).

The following definition provides a set of well-formedness constraints on SHVMs.

Definition 7 (Well-formed variability model). *A ground variability model $\mathcal{S} = M_C$ is* well-formed *if constraint ($\mathcal{S}1$) below is satisfied. A variability model $\mathcal{S} = (M_C, \{\mathit{VP}_1, \ldots, \mathit{VP}_n\})$ with variation points $\mathit{VP}_i = \{\mathcal{S}_{i,j} \mid 1 \le j \le k_i\}$ is* well-formed *if all variants $\mathcal{S}_{i,j}$ are well-formed, and furthermore, the following constraints are satisfied:*

($\mathcal{S}1$) *M_C implements artifact names at most once.*
($\mathcal{S}2$) $\mathrm{names}\,(M_C) \cap \mathrm{names}\,(\mathit{VP}_i) = \emptyset$ *for all i, and*
$\mathrm{names}\,(\mathit{VP}_{i_1}) \cap \mathrm{names}\,(\mathit{VP}_{i_2}) = \emptyset$ *whenever $i_1 \neq i_2$.*
($\mathcal{S}3$) $\mathrm{names}\,(\mathcal{S}_{i,j_1}) = \mathrm{names}\,(\mathcal{S}_{i,j_2})$ *for all i, j_1, j_2, and*
$\mathrm{impls}(\mathcal{S}_{i,j_1}) \cap \mathrm{impls}(\mathcal{S}_{i,j_2}) = \emptyset$ *whenever $j_1 \neq j_2$.*

Example 5. Consider the SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ depicted in Figure 2. $\mathcal{S}_{A1}$ is not well-formed whereas $\mathcal{S}_{A2}$ is. The separation degrees are $sd(\mathcal{S}_{A1}) = \frac{9}{6 \cdot 5} = 0.3$ and $sd(\mathcal{S}_{A2}) = \frac{9}{9} = 1$. Figure 3 depicts another two SHVMs, $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$. Neither of these are well-formed and both have separation degree $\frac{4}{5} = 0.8$.

The constraints in Definition 6 ensure that the separation degree of a well-formed SHVM is equal to one, and is thus maximal.

Proposition 1. *If variability model $\mathcal{S}$ is well-formed then $sd(\mathcal{S}) = 1$.*

Note that the converse of Proposition 1 does not hold in general: The variability model $M_C = \{a_1, a_2\}$ has separation degree 1, but well-formedness constraint ($\mathcal{S}1$) is not satisfied.

3 Relating Families and Variability Models

In this section, we present translations between well-formed variability models and simple families, and show that they are inverses of each other. In particular, this entails that the translation from simple families to variability models produces the unique well-formed model generating the respective family, thus giving a procedure for constructing a variability model from a given family.

3.1 From Variability Models to Families

The set of products generated by a ground model is the singleton set comprising the set of common artifact implementations (and, thus, representing one product). The set of products generated by a variation point is the union of the product sets generated by its variants. Finally, the set of products generated by an SHVM with a non-empty set of variation points is the set of all products consisting of the common artifact implementations and of exactly one product from the set generated by each variation point.

Definition 8 (Family generation). *The mapping $\mathit{family}(\mathcal{S})$ from variability models to families is inductively defined as follows:*

$$\mathit{family}(M_C) \overset{\mathsf{def}}{=} \{M_C\}$$
$$\mathit{family}((M_C, \{VP_1, \ldots, VP_n\})) \overset{\mathsf{def}}{=} \{M_C\} \bowtie \textstyle\prod_{1 \le i \le n} \mathit{family}(VP_i)$$
$$\mathit{where}\ \mathit{family}(VP) \overset{\mathsf{def}}{=} \textstyle\bigcup_{\mathcal{S} \in VP} \mathit{family}(\mathcal{S})$$

We say that variability model $\mathcal{S}$ generates *$\mathit{family}(\mathcal{S})$.*

Here we again abuse notation by also defining a mapping with the same name from variation points to the same co-domain. Family generation is well-defined in the sense that well-formed variability models generate simple families.

Proposition 2. *If variability model $\mathcal{S}$ is well-formed, then $\mathit{family}(\mathcal{S})$ is simple.*

Example 6. SHVMs $\mathcal{S}_{A1}$ and $\mathcal{S}_{A2}$ in Figure 2 both generate family $\mathcal{F}_A$ in Example 1, implying that family $\mathcal{F}_A$ is simple since $\mathcal{S}_{A2}$ is well-formed. SHVMs $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ in Figure 2 both generate family $\mathcal{F}_B$ in Example 1. Of these four, $\mathcal{S}_{A2}$, $\mathcal{S}_{B1}$ and $\mathcal{S}_{B2}$ have maximal separation degree in the sense that, for each of the families $\mathcal{F}_A$ and $\mathcal{F}_B$, no other SHVMs for the same family have higher separation degree.

3.2 From Families to Variability Models

We now present a reverse transformation from simple families to well-formed variability models. Recall that simple families have unique formation trees modulo commutativity and associativity of the two operations. Well-formed SHVMs can thus be seen as a uniform way of grouping the formation terms. Every family $\mathcal{F}$ can be decomposed into the form:

$$\mathcal{F} = \{P\} \bowtie \mathcal{F}_V, \quad \mathcal{F}_V = \textstyle\prod_{1 \le i \le n} \mathcal{F}_i, \quad \mathcal{F}_i = \textstyle\bigcup_{1 \le j \le k_i} \mathcal{F}_{i,j}$$

where P is a product, or equivalently, as a single equation:

$$\mathcal{F} = \{P\} \bowtie \textstyle\prod_{1 \le i \le n} \bigcup_{1 \le j \le k_i} \mathcal{F}_{i,j} \qquad (*)$$

The existence of the decomposition is ensured since every family $\mathcal{F}$ can be trivially decomposed as $\{\emptyset\} \bowtie \prod \bigcup \mathcal{F}$, *i.e.*, with product P being empty and $n = k_1 = 1$. Decomposition $(*)$ is only unique under additional constraints, under which the decomposition is called canonical.

Definition 9 (Canonical form). *A family $\mathcal{F}$, decomposed as equation $(*)$ above, is in* canonical form *if the following conditions hold:*

- *($\mathcal{C}$1) The product P is the set of artifact implementations that are common to all products in $\mathcal{F}$.*
- *($\mathcal{C}$2) The set of artifact names in $\mathcal{F}_V$ has n equivalence classes w.r.t. correlated artifact names $C^*_{\mathcal{F}_V}$, and for the i-th equivalence class, the family $\mathcal{F}_i$ is the projection of $\mathcal{F}_V$ onto the artifact names of the class.*

($\mathcal{C}3$) For all i, $1 \leq i \leq n$, $\mathcal{F}_{i,j}$ are the k_i equivalence classes of $\mathcal{F}_i$ w.r.t. implementation sharing $N^*_{\mathcal{F}_i}$.

The decomposition into canonical form is clearly unique for a simple family, and exposes one level of hierarchy. Thus, by iterative application of the decomposition, we obtain a mapping from families to hierarchical variability models.

Definition 10 (Variability model generation). *The mapping $\mathit{shvm}(\mathcal{F})$ from simple families presented in canonical form to variability models is inductively defined as follows:*

$$\mathit{shvm}(\{M_C\}) \overset{\mathsf{def}}{=} M_C$$

$$\mathit{shvm}\big(\{M_C\} \bowtie \textstyle\prod_{1\leq i\leq n} \bigcup_{1\leq j\leq k_i} \mathcal{F}_{i,j}\big) \overset{\mathsf{def}}{=} (M_C, \{\mathit{VP}_1, \ldots, \mathit{VP}_n\})$$

$$\textit{where } \mathit{VP}_i \overset{\mathsf{def}}{=} \{\mathit{shvm}(\mathcal{F}_{i,j}) \,|\, 1 \leq j \leq k_i\}$$

We say that family $\mathcal{F}$ generates *variability model $\mathit{shvm}(\mathcal{F})$.*

As the next result shows, the generated variability model is well-formed.

Proposition 3. *If family $\mathcal{F}$ is simple, then $\mathit{shvm}(\mathcal{F})$ is well-formed.*

Example 7. Consider the family $\mathcal{F}_A$ from Example 1.

- In the first step of the decomposition of $\mathcal{F}_A$ into canonical form we obtain the common set $P = \{a_1, b_1\}$ and the family $\mathcal{F}_V = \{\{c_1, d_1, e_1\}, \{c_1, d_1, e_2\}, \{c_2, d_2, e_1\}, \{c_2, d_2, e_2\},$
$\{c_2, d_3, e_1\}, \{c_2, d_3, e_2\}\}$.
- In the next step, we analyze $\mathcal{F}_V$ to find that only artifact names c and d are correlated. Projecting $\mathcal{F}_V$ onto the two resulting equivalence classes $\{c, d\}$ and $\{e\}$ we obtain the two variation points $\mathcal{F}_1 = \{\{c_1, d_1\}, \{c_2, d_2\}, \{c_2, d_3\}\}$ and $\mathcal{F}_2 = \{\{e_1\}, \{e_2\}\}$.
- In the third step, we analyze $\mathcal{F}_1$ and see that two products share the artifact implementation c_2, which gives us the variants $\mathcal{F}_{1,1} = \{\{c_1, d_1\}\}$ and $\mathcal{F}_{1,2} = \{\{c_2, d_2\}, \{c_2, d_3\}\}$, and then analyze $\mathcal{F}_2$ to obtain the variants $\mathcal{F}_{2,1} = \{\{e_1\}\}$ and $\mathcal{F}_{2,2} = \{\{e_2\}\}$.

Only $\mathcal{F}_{1,2}$ is not a ground model. Applying the above steps decomposes it into a common set $\{c_2\}$ and a single variation point with two variants consisting of the common sets $\{d_2\}$ and $\{d_3\}$. It is easy to see that $\mathit{shvm}(\mathcal{F}_A)$ is the variability model $\mathcal{S}_{A2}$ in Figure 2.

3.3 Characterization Results

Our first result establishes *correctness* of model extraction.

Lemma 1. *For every simple family $\mathcal{F}$ we have:*

$$\mathit{family}(\mathit{shvm}(\mathcal{F})) = \mathcal{F}$$

The second result establishes *uniqueness* of well-formed models w.r.t. the generated (simple) family.

Lemma 2. *For every well-formed variability model $\mathcal{S}$ we have:*

$$\mathit{shvm}(\mathit{family}(\mathcal{S})) = \mathcal{S}$$

An immediate consequence of the above two lemmata is our main characterization result, which essentially states that the two transformations relating variability models and families are inverses of each other.

Theorem 1 (Characterization Theorem). *For every simple family $\mathcal{F}$ and every well-formed variability model $\mathcal{S}$ we have:*

$$\mathit{family}(\mathcal{S}) = \mathcal{F} \iff \mathit{shvm}(\mathcal{F}) = \mathcal{S}$$

4 Application

In this section, we show how to apply our theory to families consisting of products of program code. We explain how to obtain an SHVM from a set of products, and what insights one can gain from the derived model. Our running example (Section 4.1) is a simple product family written in Java, but the application of our theory is not restricted to particular programming languages or paradigms.

4.1 Example Product Line: Storing and Processing Collections

The example family consists of six products, where each product is a Java class. The code for all products appears in Figure 4.[1] The six products—named $P_{\mathtt{X1}}$, $P_{\mathtt{X2}}$, $P_{\mathtt{Y1}}$, $P_{\mathtt{Y2}}$, $P_{\mathtt{Z1}}$, and $P_{\mathtt{Z2}}$ after the respective class—have the following commonalities: They all store a collection of values of the custom type `Elem`, have a method for setting this state to some value, a method `process()`, and last a method `compute()` which returns some subclass of `Number`. The products have the following differences: The type of the state is either `List` or `Set`, both subinterfaces of `java.util.Collection`. In the case of `List`, method `compute()` returns a `Double`, and in the case of `Set`, it returns either a `Byte` or an `Integer`. Furthermore, method `process()` either prints out the state of one element at a time using a method on class `System`, or it produces a `String` from the elements and returns it.

4.2 From Code to Artifacts

Before we can construct an SHVM, we need a scheme to obtain a set of products, that is, products in the sense of Definition 1. Thus, we must identify artifacts in the product code. An artifact name in the program code is a construct that may

[1] We have omitted the following: import declarations, definition of custom type `Elem`, and repeated or irrelevant code.

```
class X1 {
    List<Elem> state = new ArrayList<Elem>();
    void setState(List<Elem> arg) { this.state.addAll(arg); }
    Double compute() { ... }
    void process() { for (Elem e : state) System.out.println(e); }
}
class X2 {
    List<Elem> state = new ArrayList<Elem>();
    void setState(List<Elem> arg) { ... } // as before
    Double compute() { ... }
    String process() {
        String res = "";
        for (Elem e : state) res = res + "," + e.toString();
        return res;
}   }
class Y1 {
    Set<Elem> state = new HashSet<Elem>();
    void setState(Set<Elem> arg) { this.state.addAll(arg); }
    Byte compute() { ... }
    void process() { ... } // as before with same sig.
}
class Y2 {
    Set<Elem> state = new HashSet<Elem>();
    void setState(Set<Elem> arg) { ... } // as before
    Byte compute() { ... }
    String process() { ... } // as before with same sig.
}
class Z1 {
    Set<Elem> state = new HashSet<Elem>();
    void setState(Set<Elem> arg) { ... } // as before
    Integer compute() { ... }
    void process() { ... } // as before with same sig.
}
class Z2 {
    Set<Elem> state = new HashSet<Elem>();
    void setState(Set<Elem> arg) { ... } // as before
    Integer compute() { ... }
    String process() { ... } // as before with same sig.
}
```

Fig. 4. Example product line consisting of six Java classes

Table 1. Example scheme for obtaining artifacts from Java code

Art. name	*Art. impl.*	*Connection*	*Notation*
interface I	class C	C implements I	I_C
interface I	interface J	J subinterface of I	I_J
class C	class D	D subclass of C	C_D
type T	type T	(by convention)	T_T

occur several times, but with different realizations which are then the artifact implementations. Deciding how to identify artifacts in the code means determining what are the important parts of the code for the variability model of the product line. In general, this can be done in many ways. Here, we give one possible example.

For this example, we consider an artifact to be a pair of Java types, one being the name of the type and one being its implementation. For two Java types to form an artifact, they must be connected as shown in Table 1. The types that form artifacts in our example are underlined in Figure 4. (Class `java.lang.Object` and interface `Collection` do not occur in the figure, but are also used.) These are some artifacts identified in the example:

- The interface `java.util.List` is connected to the interface `java.util.Collection` via the Java implements relation, giving rise to the artifact $\texttt{Collection}_{\texttt{List}}$ (omitting package prefixes).
- The Class `java.lang.String` is connected to the class `java.lang.Object` via the subclass relation, so we have the artifact $\texttt{Object}_{\texttt{String}}$.
- Class `Elem` is—by convention—related to itself, so we have the artifact $\texttt{Elem}_{\texttt{Elem}}$.

With the scheme in Table 1, we identify the following set of products which is a simple family and yields a hierarchical variability model with three variation points including one inside the other.

$$P_{\texttt{X1}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{List}}, \texttt{Number}_{\texttt{Double}}, \texttt{Object}_{\texttt{System}}\right\}$$
$$P_{\texttt{X2}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{List}}, \texttt{Number}_{\texttt{Double}}, \texttt{Object}_{\texttt{String}}\right\}$$
$$P_{\texttt{Y1}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{Set}}, \texttt{Number}_{\texttt{Byte}}, \texttt{Object}_{\texttt{System}}\right\}$$
$$P_{\texttt{Y2}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{Set}}, \texttt{Number}_{\texttt{Byte}}, \texttt{Object}_{\texttt{String}}\right\}$$
$$P_{\texttt{Z1}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{Set}}, \texttt{Number}_{\texttt{Integer}}, \texttt{Object}_{\texttt{System}}\right\}$$
$$P_{\texttt{Z2}} = \left\{\texttt{Elem}_{\texttt{Elem}}, \texttt{Collection}_{\texttt{Set}}, \texttt{Number}_{\texttt{Integer}}, \texttt{Object}_{\texttt{String}}\right\}$$

4.3 Constructing and Interpreting the SHVM

From the set of products obtained in the previous section, constructing an SHVM is straightforward by the procedure specified in Definition 10. We obtain the SHVM depicted in Figure 5. The SHVM in this figure is nearly identical to $\mathcal{S}_{A2}$ in Figure 2—differing only in the cardinality of set at the leftmost branch from the root. Hence, the construction proceeds similarly to that of Example 7. Since the family is simple, the obtained model is well-formed and, thus, optimal w.r.t. the separation degree.

The constructed SHVM may be read as a graphical summary of the textual product line description given in Section 4.1, focusing on Java types. Note, in particular, that the choice between `List` and `Set` is clearly visible as a variation point, and that, for example, the combination of `List` and `Byte` is not allowed by the SHVM, whereas `List` and `Double` is allowed.

5 Related Work

The existing approaches to represent solution space product line variability can be divided into three directions [23]. First, annotative approaches consider one

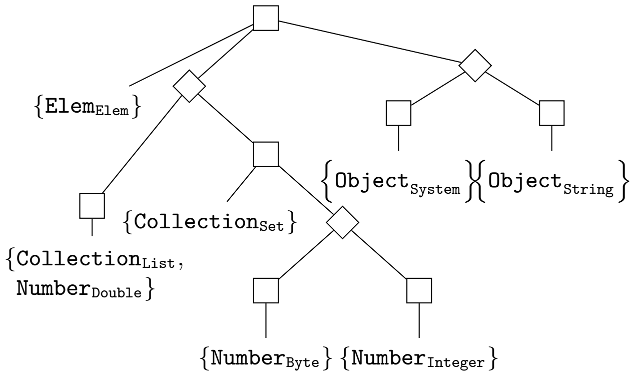


Fig. 5. SHVM for the example family

model representing all products of a product line. Variant annotations, *e.g.*, using UML stereotypes [24,10], presence conditions [6], or separate variability representations, such as orthogonal variability models [18], define which parts of the model have to be removed to generate the model of a concrete product. Second, compositional approaches [4,23,16,3] associate product fragments with product features which are composed for particular feature configurations. Third, transformational approaches [13,5] represent variability by rules determining how a base model has to be changed for a particular product model. All these approaches consider a representation of artifact variability without any hierarchy.

Our hierarchical variability model generalizes the ideas of the Koala component model [22] for the implementation of variant-rich component-based systems. In Koala, the variability of a component is described by the variability of its sub-components which can be selected by *switches* and explicit *diversity interfaces*. Diversity interfaces and switches in Koala can be understood as concrete language constructs targeted at the implementation level to express variation points and associated variants. Plastic partial components [17] are an architectural modeling approach where component variability is defined by extending partially defined components with variation points and associated variants. However, variants cannot contain variable components so this modeling approach is not truly hierarchical. Hierarchical variability modeling for software architectures [12] applies the modeling concepts for solution space variability presented in this paper to component-based software engineering and provides a concrete modeling language for variable software architectures that is truly hierarchical. However, none of these approaches formally defines the semantics of hierarchical variability models, nor reasons about their well-formedness or uniqueness.

To the best of our knowledge, this paper presents the first approach for constructing a hierarchical variability model for solution space variability from a given product family. So far, there have only been approaches to construct feature models for representing problem space variability for a given set of products. Czarnecki *et al.* [8] re-construct a feature model from a set of sample feature combinations using data mining techniques [1]. Other approaches aim at constructing feature models from sample mappings between products and their features using formal concept analysis [9], for instance, to derive logical dependencies between code variants from pre-processor annotations [21], or to construct a feature model for function-block based systems after determining model variants by similarity [19]. Loesch and Ploedereder [14] use formal concept analysis to optimize feature models in case of product line evolution, *e.g.*, to remove unused features or to combine features that always occur together. Niu and Easterbrook [15] apply formal concept analysis to functional and non-functional product line requirements in order to construct a feature model as a more abstract representation of the requirements. Also, information retrieval techniques are applied to obtain a feature model from heterogeneous product line requirements [2]. Using hierarchical clustering, a tree structure of textually similar requirements is constructed. Requirement clusters in the leaves are more similar to each other than requirements clusters closer to the root giving rise to the structure of a feature model.

In our work, we abstract from the need to determine the different variants of the same conceptual entity by assuming fixed artifact names and corresponding artifact implementations. However, if we relax this assumption, techniques, such as similarity analysis [19] or formal concept analysis [9] could be applied to infer the relationship between different variants of the same conceptual entity, and thus make our approach applicable.

6 Conclusion

In this article, we present hierarchical solution space variability models for software product lines. We give a formal semantics of such models in terms of sets (or families) of products, where each product is a set of artifact implementations. We introduce the separation degree as a quality measure of hierarchical variability models. We identify well-formed variability models as a class of models for which the measure is maximal (and equal to one) and which are unique for the family they generate; the class of families generated by such models is the class of simple families. Furthermore, we present a transformation that constructs, from a simple family, the unique well-formed model that generates it, and prove uniqueness by showing that family generation and model construction are inverses of each other for this class of models. While maximal separation degree and uniqueness of models with maximal measure are theoretically appealing, in practice, product families might not be simple. Still, the separation degree is a useful measure for hierarchical variability models, and, as Examples 5 and 6 suggest, searching for the set of models with a maximal measure (not necessarily equal to one) for a given family is equally meaningful.

Future work will focus on the practical evaluation of the proposed method for variability model mining, considering in particular sets of (legacy code) products that have not been designed as a family from the outset. Further effort is planned on generalizing the model with optional and multiple variant selections and with requires/excludes constraints between variants, and on adapting accordingly the model reconstruction transformation. Another generalization will deal with the more abstract domain of products over implementations only, where the names are not given in advance, but must be inferred. Additionally, the restriction that all variants associated to a variation point have to provide the same artifact names will be lifted. In order to integrate hierarchical variability models into software product line engineering, we aim at offering tool support for hierarchical variability modeling extending the approach presented in [12] and connecting variation points to product features captured by feature models.

References

1. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In: SIGMOD Conference, pp. 207–216 (1993)
2. Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., Rummler, A.: An exploratory study of information retrieval techniques in domain analysis. In: SPLC, pp. 67–76 (2008)
3. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model Superimposition in Software Product Lines. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 4–19. Springer, Heidelberg (2009)
4. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Eng. 30(6), 355–371 (2004)
5. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. In: GPCE. Springer (2010)
6. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
7. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
8. Czarnecki, K., She, S., Wasowski, A.: Sample spaces and feature models: There and back again. In: SPLC, pp. 22–31 (2008)
9. Ganter, B., Wille, R.: Formal Concept Analysis: Mathematical Foundations. Springer (1996)
10. Gomaa, H.: Designing Software Product Lines with UML. Addison Wesley (2004)
11. Gurov, D., Østvold, B.M., Schaefer, I.: A hierarchical variablility model for software product lines. Technical Report TRITA-CSC-TCS 2011:1, KTH Royal Institute of Technology, Stockholm, 26 pages (2011), `http://www.csc.kth.se/~dilian/Papers/techrep-11-1.pdf`
12. Haber, A., Rendel, H., Rumpe, B., Schaefer, I., van der Linden, F.: Hierarchical variability modeling for software architectures. In: Software Product Line Conference, SPLC 2011 (2011) (to appear)
13. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In: Software Product Line Conference (SPLC 2008), pp. 139–148. IEEE (2008)
14. Loesch, F., Ploedereder, E.: Optimization of variability in software product lines. In: SPLC, pp. 151–162 (2007)

15. Niu, N., Easterbrook, S.: Concept analysis for product line requirements. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development, AOSD 2009, pp. 137–148 (2009)
16. Noda, N., Kishi, T.: Aspect-Oriented Modeling for Variability Management. In: Software Product Line Conference (SPLC 2008), pp. 213–222. IEEE (2008)
17. Pérez, J., Díaz, J., Soria, C.C., Garbajosa, J.: Plastic Partial Components: A solution to support variability in architectural components. In: WICSA/ECSA, pp. 221–230 (2009)
18. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer (2005)
19. Ryssel, U., Ploennigs, J., Kabitzsch, K.: Automatic variation-point identification in function-block-based models. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, pp. 23–32. ACM, New York (2010)
20. Schaefer, I., Gurov, D., Soleimanifard, S.: Compositional Algorithmic Verification of Software Product Lines. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 184–203. Springer, Heidelberg (2011)
21. Snelting, G.: Reengineering of configurations based on mathematical concept analysis. ACM Trans. Softw. Eng. Methodol. 5, 146–189 (1996)
22. van Ommering, R.: Software reuse in product populations. IEEE Trans. Software Eng. 31(7), 537–550 (2005)
23. Völter, M., Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In: Software Product Line Conference (SPLC 2007), pp. 233–242. IEEE (2007)
24. Ziadi, T., Hëlouët, L., Jézéquel, J.-M.: Towards a UML Profile for Software Product Lines. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 129–139. Springer, Heidelberg (2004)